FPGA Flight Controller

Jack Williams - s3601511

1. Summary

This report details the development and evaluation of a flight controller system with digital hardware acceleration to improve performance and simplify software.
First the report will introduce the problems with existing flight controller designs, followed by how the use of a Field Programmable Gate Array can improve the situation. Finally it will detail the issues obstructing an FPGA implementation.
As the first step in the design, a literature review is conducted to establish how others have implemented similar systems. Once this information is distilled, the development methodology is established. Including the tools and workflow necessary for each design component (or module).
After this, an overview of the system architecture will be shown, with an explanation of the function of each module and how the design was altered during testing. The design process and results will be discussed with a critical evaluation of the method and its outcomes.
Subsequently, a performance evaluation will be conducted on the new system. Which will establish the success of the design from a performance perspective.
Finally, the report will conclude with some recommendations for future work.

2. Introduction

Typically, a flight controller is defined as a system, usually electronic, that assists or allows an airborne vehicle to control its position and attitude while in flight.
In the particular field of multirotor UAVs, a flight controller is required to maintain stability. Pilot inputs are simply used to modify the goal state that the stability system aims for. For example, the pilot might request a desired attitude or speed through the transmitter. Like some modern aircraft, these multirotors are unflyable without the assistance of a computer [1].

For research applications, these flight controllers require a combination of performance, flexibility and simplicity. And should allow the easy implementation of new airframe configurations, control system software or sensor inputs. While having enough spare performance to cope with the extra workload. In flight controller terms “performance” is usually defined as the stability of the aircraft in regards to its goal state. For example, how much does a particular disturbance affect the aircrafts state? A more performant system will return to the desired state more quickly and accurately.
The performance of a system is affected by how quickly it can feed sensor inputs into its internal model and produce a response. As well as how accurately the internal model responds. The delay between data being fed in and acted upon is a measurable performance metric and will be the focus of this report.

2.1 Gap Analysis

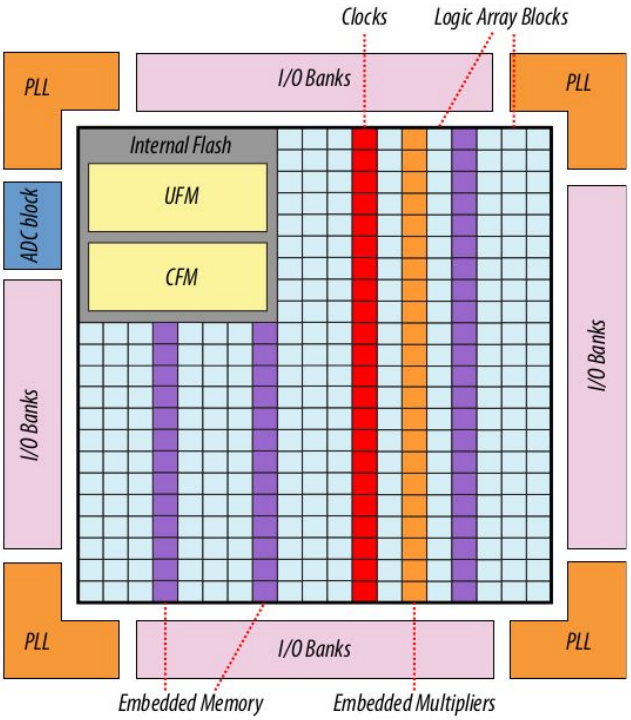
There are a few reasons why conventional approaches to flight controller design have fundamentally compromised performance.
The conventional approach to multirotor flight controller hardware is through the use of an SOC, or system on chip. Which implements a large sequential computer in a single small package. The hardware at work in these complex systems are usually powerful microcontrollers, like the current ARM Cortex series [2].
The multifaceted task of managing sensor, commands, stability and output operations can be challenging for a single processor. Especially with some of the intensive mathematical operations involved in maintaining stability, such as numerical integration and kalman filtering [3]. When the processor does not have sufficient resources it will cause the system latency to increase, and hence overall flight controller performance to decrease.

In addition, RMIT requires an accessible platform for multirotor research projects. The management of the intensive tasks discussed above can lead flight controller designers to large, complex and slow flight control software. Which has to balance numerous real time tasks with multiple priorities. Typically this complex software can become an obstacle when repurposing the system. Increasing development time.

But other flight controller hardware options exist that can ease this complexity while also relieving associated performance issues, this project shall be exploring one in particular.

2.2 FPGAs as the Solution

Field Programmable Gate Arrays, or FPGAs, consist of many discrete digital logic devices, such as gates, memory and look up tables. This hardware can be interconnected through software, to create more complex digital systems, such as a conventional processor. The flexibility of this system allows the creation of very specific and optimised hardware for niche applications.
Hence FPGAs can act somewhat like an Application Specific Integrated Circuit, or ASIC. Except without the large costs associated with ASIC development and tooling.
Below can be seen an FPGA schematic, consisting of (among other things) many thousands of discrete logic components.



FPGA Diagram - Intel, "MAX 10 FPGA Device Architecture," p. 31. [4]

FPGAs allow industries and applications that might otherwise not have the resources for ASIC development to access the performance of custom silicon. However the performance when compared to ASICs will always be comparatively worse, as FPGAs incur a performance handicap for their complex internals and subsequent flexibility.

In respect to UAV flight controllers, this facilitates moving some of its tasks into hardware modules within the FPGA. Which has the potential to:

- Improve latency and update frequency
- Reduce the size and complexity of the codebase
- Make it easier for new features to be added to existing hardware

FPGAs do have limitations on the amount of digital logic that can be implemented on one unit. The usage of the internal resources shall be referred to as resource usage for the context of this report. The resource usage of an FPGA is greatly affected by the type of the logic implemented, so methods for precalculating the resource usage of a design are not used.

2.3 Problem Statement

Naturally an FPGA implementation comes with its own set of problems, this report aims to address each of them, and produce a working example to demonstrate the designs effectiveness.

2.3.1 Design

However, many of the design decisions surrounding an FPGA flight controller implementation are not straightforward, as the sheer flexibility of the hardware can overwhelm any attempts to pin down system details.
Many architecture decisions are needed to lay out interfaces and system boundaries, which will allow the work to be divided into individually manageable portions.
Much of this design will be drawn from historical analysis, identified in the Literature Review (to follow). The final design will be discussed in the Development Results section.

2.3.2 Construction

Furthermore, the creation and testing of FPGA “programs” requires specialist knowledge and experience with a suite of software tools unique to the platform. In order to complete the design implementation, a Development Methodology will be constructed detailing the tools and processes necessary to implement the design.
Testing and debugging of FPGA projects requires further work and a thorough process in order to combat the oftentimes obstructive tools.

2.3.3 Results

In order to draw conclusions from the project, a quantifiable performance metric and metric measuring system must be established, this will be discussed in the Performance Evaluation section.

3. Literature Review

Research papers provide few examples of FPGA flight controller implementations. However, from the few that do exist, important information can be extracted about how others have utilised the FPGAs flexibility.
Each of the following sections lists a design factor, and each historical examples approach will be subdivided by their approach to this design factor.

3.1 Overall Architecture Types

Of particular interest to this report is exactly what tasks the FPGA fulfils within these flight control systems. As such an innately flexible piece of hardware, analysing historical configurations may help to understand the design space, as well as the compromises and tradeoffs made in each design. The following sections each describe a different architecture type that was seen in previous implementations of FPGA flight controllers.

3.1.1 Hardware Sensor Processing Only

The only common feature between each of the five implementations is that they utilise the FPGA for processing the input data from sensors. In one extreme instance [5], the FPGA

was only used for processing inputs, and flight control was done on a seperate microcontroller connected to the FPGA via serial connection.
This makes it clear that the parallel nature of FPGAs is suited toward the independent collection of disparate data from sensor sources. Which “lets the CPU concentrate on controlling the aircraft” [6], as some data collection tasks can be expensive and interrupt the CPU during real time calculations.

3.1.2 Entirely Hardware Implementation

Implementations [7] and [8] take FPGA utilisation to the other end of the spectrum, and have no external CPU or softcore at all. Stabilisation is performed using closed loop control embedded directly in the FPGA.
Unfortunately, neither paper focuses on the performance of such an approach, and both stabilisation methods are extremely basic by modern standards. Furthermore, neither hardware actually flew, with [8] confined to a simulation and [7] only demonstrated on a test stand.
While this design choice might offer theoretically minimal latency, it appears that implementation of a complex modern control system inside the FPGA fabric may not have been worth the performance improvements. Furthermore, the decrease in flexibility for what is intended to be easy to modify system makes this sort of design more unappealing.

3.1.3 Accelerated Stability Calculation

In paper [6], a middle ground between the stability coprocessor of [5] and stability hardware modules of [7] and [8] is taken.
In this approach a softcore CPU offloads input, output, and select intensive stability calculation tasks to FPGA hardware modules. While the control loop is still run sequentially, it allows (for this implementation) a minimum of only 11 CPU instructions to be run each loop. Despite the potential performance improvements, the paper still touts that “the design is easily expanded using hardware modules or by modifying the control software”[6].

This architecture type is selected for further consideration.

3.1.4 Accelerated Cascaded Speed Control

One area where improvement in control latency has been suggested to have a significant impact is the translation of propeller speed to motor throttle.
This proposal stands out in practicality because the relationship between propeller speed and thrust is almost linear for conventional flight [9]. This means the system can be modelled accurately with simple PID control.
Systems that control the motor throttle in a separate control system to the attitude, known as “cascaded motor speed control and attitude control” systems, have already been shown to improve hovering stability [10].
With a simple implementation that can be generalised among many different multirotor configurations, a cascaded speed control is an effective way to leverage the abilities of FPGA hardware with minimal development resources.

But it creates issues regarding ease of configuration in the event that the control constants (ie PID gains) of the system change.

3.2 IO Interface Design

Another area of interest is the interface between hardware IO, such as sensors, and sequential processor modules (if any).
The possible designs for this bus are numerous and the advantages and disadvantages of each are not immediately apparent. RMIT also wishes to avoid interrupt driven systems due to the added complexity.

Few of the references go into much detail on this system, referring to it only as a “concurrent bus” [11]. Only implementation [6] has enough information to be of use.
In this implementation, sensor data is accessed when the CPU receives an interrupt from the hardware informing it that new data is available. It is then occupied for 3 clock cycles as it transfers the data into local registers.
This method has minimal CPU load when compared with conventional flight controllers, but could still potentially be made even less CPU intensive through direct memory access or similar.

Unfortunately all the historical FPGA flight controller examples found described some form of interrupt driven IO mechanism. As such, caution must be used in the IO systems design.

3.3 Sensor Initialisation

One gap in the current research centers around identifying ways of dealing with the initialisation of peripherals.
The problem of initialisation is identified in paper [7], where special modifications had to be made to the hardware controlling the ESC output to initialise at system start. The issue is created when hardware modules are used to manage IO, and one or more peripherals require configuration data to be sent in order to start operating.
With more complex peripherals, creating hardware modules to configure the external devices would become significantly more complex. Additionally, given the sequential nature of serial communication, the parallel nature of FPGA programming does not lend itself well to this task.
Furthermore, flight controller software designers would benefit from having control over each device configuration without having to reprogram the FPGAs hardware modules (a task requiring a large toolchain and specific knowledge).

In order to minimise complexity while still maintaining performance. Hardware functions that allow the passing of peripheral control from CPU to hardware module will be investigated as a solution to all these problems. When the system starts, the CPU could configure and setup each peripheral sequentially, and then pass communication over to the hardware modules designed to communicate with that device.
If the solution works as intended, it would increase the time taken for the system to start, but otherwise still maintain the FPGAs potential performance advantage.

4. Development

4.1 Methodology

4.1.1 Tools

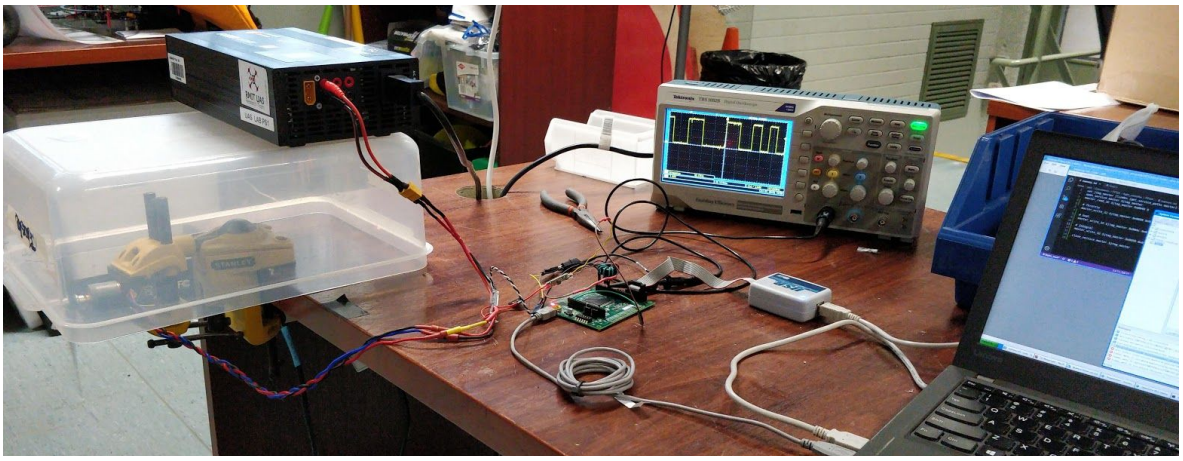
The FPGA product ecosystem was chosen based on what RMIT already had resources for. Consequently an Intel/Altera FPGA development board and associated Quartus software were selected as the testbed for the project.

Testbench/Hardware

Once the FPGA is programmed, testing with motors and other associated hardware is essential in order to debug and tune. The FPGA MAX10 development board connects to a USB JTAG adaptor that allows it to be accessed via the laptop. The board is powered over another USB connection and interfaces with hardware via a GPIO header.
The oscilloscope can then be used to monitor signals into and out of the FPGA, ensuring they behave as expected.
The electric motor is securely mounted with a vice and covered with a clear plastic container to limit exposure to spinning propellers.
A UART serial adapter is used to receive data streams from the FPGA for logging and further analysis.

Components

- FPGA Development Board
- USB Blaster
- Oscilloscope
- Multimeter
- Motor + Mounting
- ESC + Speed Sensor
- Power Supply
- UART Serial Adapter
- IMU
- Laptop



FPGA and Motor Testbench

Software

FPGAs “programs” are files that specify a configuration of the FPGAs digital logic components, ie a description of how the FPGA is configured for a particular application. These binary files are sent over to the development board via a JTAG adaptor, and are created through Intel/Alteras integrated development environment Quartus.

Within Quartus the program is created in the hardware description language Verilog. Which can be compiled, or “synthesized” into an arrangement of the FPGAs digital components. Verilog is an interesting language due to its concurrency, any portion of the program can be in “execution” at any time. This can make it difficult for those familiar with sequential programming languages to formulate their ideas correctly.

Quartus contains a library of premade digital components, anything from simple arithmetic units up to reasonably full featured CPUs. It can also generate different types of bus interfaces to connect these components automatically through a tool called Qsys. Many of the components and buses are proprietary, but the user can add custom components (perhaps designed through Verilog as above) to take advantage of Qsys for their particular application.

Also included in the Quartus software is a simulation tool called Modelsim, which allows the designer to test Verilog programs locally. This tool is a key component of the workflow, as information about the FPGAs internal state can be difficult to obtain from the development board, but very easy through the simulator.
The user may also find the need to create testbenches to automatically configure the Verilog module for simulation. As often the configuration of the modules input and outputs may differ if the module is running in a simulated environment, or among other modules in hardware.

Quartus also comes with the toolchain for building applications to run on the premade CPU mentioned earlier. This CPU, the NIOS II, can be detected over the JTAG interface and flashed with a program via the Nios Software Build Tools. The SBT compiles the CPU code, creates the Operating System, programs the processor over JTAG and sets up a remote GDB instance for debugging the software in realtime.

The CPU code for this project was written in C++, an object oriented language that can make it difficult to interact with the hardware on the bit by bit level required, but allows the abstraction of more complex flight controller concepts and is reasonably approachable.

In addition to memory interactions done through the CPU, Quartus can access certain hardware functions via a command line interface. This interface uses the TCL scripting language to automate JTAG interface functions, such as accessing mapped memory. Hardware modules in the FPGA that are connected to the built in Avalon Memory Mapped bus can be accessed via TCL functions as long as each register or memory location has been assigned an address. Management of the Avalon bus is done through Qsys (mentioned above).

In this application, post processing of the raw register data was needed in order to aid in tuning some of the hardware modules. This script was written in TCL and will be discussed in more detail in the Development Results section.

Microsoft excel was also used to graph and analyse motor speed data collected during experiments.

4.1.2 Development Process

Developing the architecture required research and planning to establish the design, and then further planning and research when the development inevitably ran into problems with integration.

The development process starts with the design of the artitecture:

- Literature review/historical analysis
- System requirements developed
- System architecture/diagram
    - Deciding upon boundaries between system components
- Component requirements
    - Which other components need to be connected and in what way
    - Debugging methods
- Integration
    - For some Quartus problems removing every component and adding them back in was required

Once the system is designed. The development of the custom FPGA modules begins:

- Module declaration written, declaring inputs and outputs along with their sizes
- First implementation, correcting errors until it compiles without any severe warnings
- Writing testbenches to automate simulation, initialising variables etc.
- Simulation in Modelsim
    - Can produce major issues that require a second pass
- Hardware test,
    - Using the oscilloscope and multimeter to gather more information about hardware outputs.

- Inevitably more simulation to further clarify the issues seen under hardware testing
- Final pass to correct these issues

Other premade components might be built in to Quartus, and vary on a case by case basis.

4.2 Results

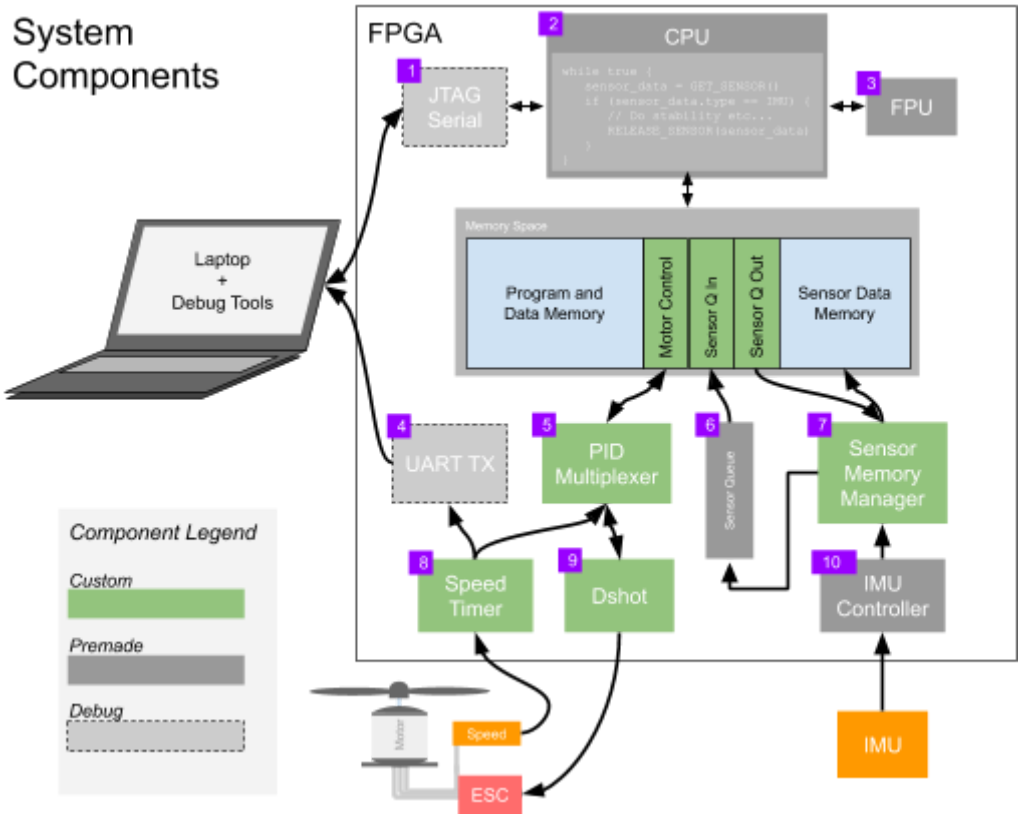
4.2.1 Architecture

From the literature above, a system design with a sequential processor and targeted hardware acceleration was chosen. Most of the initial design decisions revolved around what flight controller tasks could reasonably be transferred over to hardware implementations. From the literature, the following list of hardware implementable flight controller tasks emerged:

- Cascaded speed control PID loop calculation
- Sensor data management
- Added CPU instructions that facilitate:
    - Filtering of sensor data
    - Acceleration of CPU PID tasks

After an assessment of the work required for each option, each kind of hardware acceleration was prioritised. With cascaded loop control and hardware sensor management making essential pieces of the system, they were scheduled to be implemented first. Whereas added CPU instructions for filtering and PID operations are not required for base functionality and hence have a lower priority.

The final architecture is shown below. It describes a central CPU running flight controller code connected to sensors and outputs via memory mapped interfaces. The layout was implemented using the Qsys tool.
Each of the components will be covered in the Components section of further below.
The sensor data management system will be covered in the Sensor Memory Manager component.



The following three sections are based off both color and component number in the diagram above. The component number is in the title of each component.

4.2.2 Custom Components

Green on the System Components Diagram.
These are components implemented from a blank Verilog text file.

Speed Pulse Timer (8)

The speed sensor produces a pulse every time the motor completes a unit of rotation. This module counts the clock cycles in between pulses from the speed sensor, and then shares them with the PID system over a bus. This allows the system to infer the relative speed at which the motor is rotating (a unit of rotation can vary depending on the design of the motor).

It was found through testing that the sensor input could not be read as high or low with 100% accuracy. Which sometimes caused significantly incorrect speed values to be passed on to the rest of the system.

To resolve this problem, a filter was implemented which required the sensor readings over five sequential clock cycles to be the same before the value was confirmed as either high or low.

PID Multiplexer (5)

Background

A PID or Proportional Integral Derivative controller takes inputs regarding a systems current and desired states, and controls the system to efficiently reach the goal state. In this application, the PID controller accepts goal and current rotation speeds, and directs the throttle of the motor to achieve the goal rotation speed.

Design

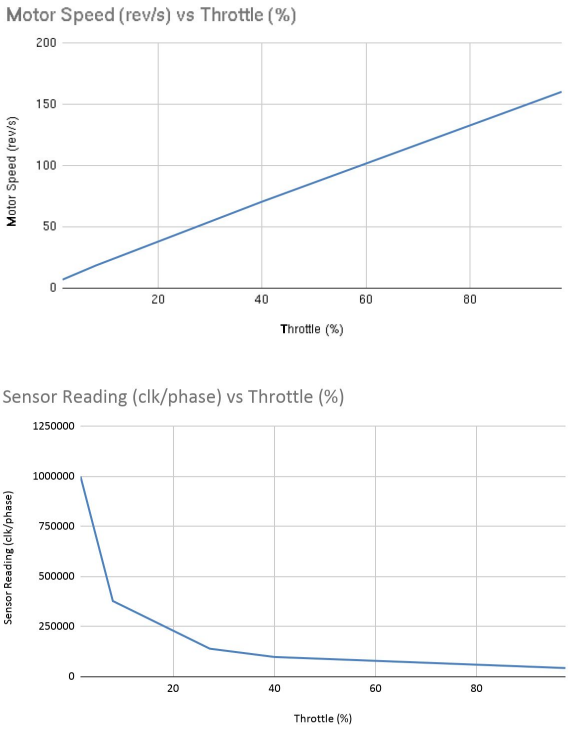
This module determines the throttle value according to a PID controller every clock cycle. Its gains and registers are variable, which allows it to service a different motor every clock cycle. The module can be shared without any loss of performance because the communication between motor and FPGA only allows updating of throttle values every few thousand clocks, so faster updates are redundant. To this end, the PID calculation is restricted to clocks where the Dshot output module has requested a new throttle value.
It also facilitates control of the PID variables and motor system settings through a memory mapped interface. And uses overflow protected signed arithmetic functions to supplement the lacking native Verilog capability.
The Dshot throttle protocol reserves the first 47 throttle values for ESC commands (such as reverse rotation), and provides 2000 units of throttle input [12]. The PID module trims and offsets the throttle values accordingly.

Testing

During debugging difficulty was encountered understanding the modules behaviour during conditions that would have been very time consuming to simulate. To further understanding of the modules internal state, some internal registers were memory mapped.

Unfortunately difficulty was then encountered tuning and debugging the system due to the effort of reading and writing the expanding catalogue of PID variables. Which were only displayed as hexadecimal memory values using the JTAG memory access tool.
A tcl script was created in the Quartus system console to parse the hexadecimal register values into human readable signed decimal with labels indicating the variable (eg. Proportional Gain).

The accuracy of the PID module was severely affected by the nonlinearity of the speed signal. Which reads data in FPGA clock cycles per motor phase revolution.
This value can be converted into motor revolutions per second by simply dividing by the clock speed and number of motor phases. However, because division operations have extremely high FPGA resource usage they were initially avoided in the hope that the nonlinearity of the raw speed data would not be enough to render the PID system ineffective.



A comparison of speed signal data with and without inverting it

However during testing the system was deemed to require the inversion from clk/phase to phase/s. And a hardware divider was added to the system.

Later on, a feedforward term was added during testing due to the constant throttle signal required to maintain a constant motor RPM. The purpose of the feedforward term is simply to calculate steady state throttle required using the linear relationship between motor speed and throttle percentage (visible in the left graph above) and add it to the output signal. The lack of a feedforward term in the initial design was an oversight.

Some hardware bugs still remain in the PID module, large errors can cause mathematically incorrect throttle values to be calculated. This problem does not affect the results gathered.

Dshot Output Module (9)

Background

Dshot is an electric motor speed controller protocol that conveys digital ESC commands over PWM, using one pulse length for logical 1 and one for a logical 0. The protocol centers around communication of the throttle values. Which under normal operation comprises 11 bits of the 16 bit Dshot packet. The remaining bits are a checksum and request for ESC telemetry data (over another theoretical communication channel). [12]

Design

The Dshot module simply repeats the 12 bits of data sent to it via a bus in the Dshot protocol, and calculates the 4 bit crc. Upon finishing one packet, it communicates to the PID system that a new throttle value is required.

Testing

Despite measurement of the timing results using an oscilloscope, during testing the timings used according to the protocol specification did not work.
A known working system using the betaflight implementation referenced earlier [12] was then used to reverse engineer the correct timings.
It was found that the ESC required an extended logical 1 pulse, additionally the gap between Dshot packets had to be extended to over 10 times its original value.

Sensor Memory Manager (7)

Background

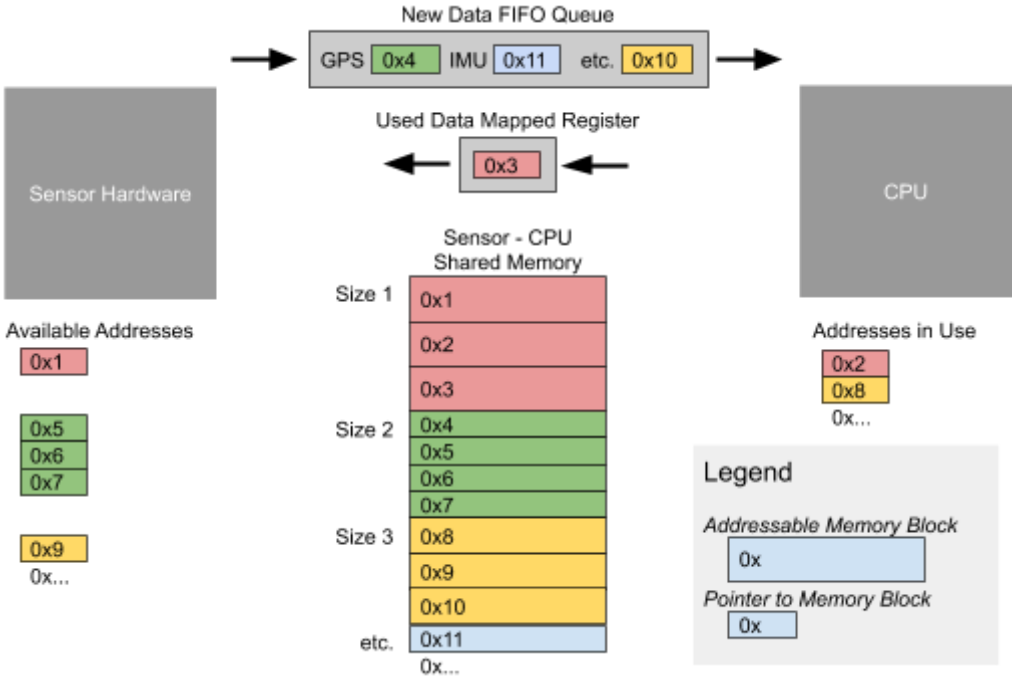
Sensor data coming into the system does not get written in a single clock, but usually comes in small parts that have to be collected together to form useful information. Additionally, communication with the CPU is not predictable, as the CPU may be busy and unable to receive information from a sensor at the exact moment it is available. The CPU may also need to access multiple copies of data from the same sensor in any order.

The sensor memory manager is designed to organise incoming sensor information for later access via the CPU. It managers the sensor memory to allow multiple iterations of sensor data to be stored simultaneously and the memory reused once the CPU has finished with the data.

Design

The sensor memory manager development process was more in depth due to the lack of historical examples.
The design operates based on the passing of memory base addresses between the CPU and the hardware. When the sensor data is received by the hardware, it writes it to a section of memory shared between it and the CPU.
Because the sensor values are received 8 bits at a time. One port allows writes with a width of 8 bits, and the other port facilitates reading with a width of 32 bits. When the sensor has finished writing its message, it then inserts the address at which the new data is located and the sensor that wrote the data into a First In First Out queue created from premade FPGA software. The hardware then records that memory as no longer available for further sensor data.
The CPU reads the information from the FIFO queue and carries out whatever tasks are associated with that sensor. Once it is finished with the data, it writes the memory address back into another buffer, where it is read by the hardware. Once the hardware receives this address, it makes that sensor memory available for writing once more.

This mechanism allows the system the flexibility to keep as many old copies of sensor data as the CPU requires, and also allows the CPU to interpret the new sensor information whenever its ready.



Testing

Although construction was finished, no testing was able to be done on this module. This was down to disorganisation (discussed later in section 4).

4.2.3 Premade Components

Gray on the System Components Diagram.
These components are provided with the tools or have come from other RMIT projects.

Sensor Queue (6)

Acts as a buffer for the Sensor Memory Manager output. Allowing the CPU to read from the queue when it's ready. Reading from the queue will remove the entry and replace it with the next one.

FPU (3)

Floating Point Unit, a coprocessor that allows the CPU to do floating point arithmetic significantly faster. Appropriate for this design due to the exclusive use of floats for the RMIT flight controller code.

CPU (2)

A design from the FPGA manufacturer, different features can be enabled and disabled for optimized resource usage. The CPU runs flight control software developed in a previous RMIT project. With alterations to support the sensor data mechanism described in component 7, and drivers added to allow easy access to hardware functions.

4.2.4 Debug Components

Light gray with dotted lines on the System Components Diagram.
These components are used for debugging and tuning and are not critical parts.

UART TX (4)

This module receives 32 bit speed sensor packets from the Speed Pulse Timer module and transmits it over a 115200 baud rate UART connection 8 bits at a time. It has a buffer of 1024 8 bit segments, in an attempt to allow it to operate for short periods of time at higher RPMs. However at some higher RPMs the module does not have enough bandwidth to maintain a real time data output.

During testing it was apparent that some of the data was corrupted, however this happened sporadically enough that it was not deemed necessary to slow the baud rate.

JTAG Serial (1)

Sourced from the Altera IP catalogue. This module interfaces with the CPU to allow it to forward a serial connection across the JTAG link. This is used as a console in the current flight controller software, allowing printf() statements to output text in a useful location.

4.2.5 Debug Tools (On Laptop)

Tuning TCL Script

This tool parses the hexadecimal format register values from the PID module via the Quartus command line interface (discussed in the Tools section). It formats each register with a label denoting what its value represents.
Additionally it formats some of the more important PID tuning values into equations that help the user determine why the system is producing the current output. For these it parses the hexadecimal into signed and unsigned number values.
TCL has no native signed hexadecimal parsing capability, so the work “Negative” was used to represent negative values. Furthermore, the reading of the register values from memory does not happen within one FPGA clock cycle, so the values are not guaranteed to sum correctly.

```
% source /home/jack/Documents/FPGA/flight_
P_gain: 0x00000001
I_gain: 0x00000001
D_gain: 0x00000001
goal: 0x7fffffff
integral_max: 0x000007d0
integral_min: 0xfffff830
throttle_override: 0x00000000
integral: 0x00000000
integral_shift: 0x0000000d
P: 0xffe00000
proportional_shift: 0x0000000a
rotation_interval: 0x00000000
output_throttle: 0x00000000
I: 0x00000000
D: 0xff000000
throttle: 0xff1fffff
derivative_shift: 0x00000007
F_gain: 0x00000001
feed_forward_shift: 0x00000009
F: 0x003fffff

Error: 0/2147483647 -2147483647

P: -2147483647*1>>10 = Negative
I: 0*1>>13 = 0
D: -2147483647-preverror>>7 = Negative
F: 2147483647*1>>9 = 4194303

Throttle: Negative->0
```

TCL Script Sample Output

4.3 Discussion

Not all planned testing was completed, several explanations have been identified for this. Many of the tools involved with FPGA development, parts of the Quartus software, are not either intuitive or well maintained. Which can lead to irritating issues necessitating trial and error problem solving:

- Qsys does not support scientific notation in parameters and interprets them as strings.
- Qsys interface fails to configure the memory IP blocks in uneven dual port sizes (sensor memory).
- Qsys causes errors in the CPU initialization/bootloader if the CPU name is set to “CPU”.

Verilog, the hardware description language used to write the custom system components, can also be difficult to understand. The more time consuming issues with the Verilog language were:

- Saturation arithmetic is not natively supported in Verilog and creates difficult to read code when implemented manually.
- Both inputs and outputs of a Verilog arithmetic operation must be signed for the operation to perform signed arithmetic correctly.

In future, these scheduling issues can be alleviated by understanding the comparative size of the tools and the size of the industry that maintains them. If the tool is large and complex and the industry relatively small, it can be assumed that the tool will have usability issues that may increase the time taken unnecessarily. More caution should be shown when estimating project time frames without experience in the tools.

Another issue that arises from lack of tool experience is the quality of the result. Some of the modules with synchronous state changes, in particular the Dshot module, are difficult to read or debug. This is because priority was given to creating programs with a small amount of variables and lines of code. This approach can create more readable and performant programs in interpreted languages like Python. But in Verilog a more explicit style tends to create faster and more readable programs.

The project is currently resource limited, meaning the amount of logic units available on the FPGA has restricted the amount of functionality that can be implemented.
One way that Verilog programming is similar to conventional programming is that declaring operands as constants rather than variables can have resource utilisation benefits. In its current state the PID module has many variables and inputs that can be made constant when their values have been determined through experimentation.
Additionally, resource usage could be further minimized by decreasing the capabilities of the inbuilt CPU, which can be configured in a very granular fashion through Qsys.

Finally, the last minute feedforward term addition did waste some of the allotted testing time on an issue that was down to a fundamental flaw in the design rather than an integration or tuning problem. More care should have been taken during the research phase to ensure the design could theoretically perform as required.

5. Performance Evaluation

To verify that the implemented FPGA hardware had the potential to improve multirotor performance. An experiment was conducted with and without the inbuilt PID hardware module, and the motor response was observed.

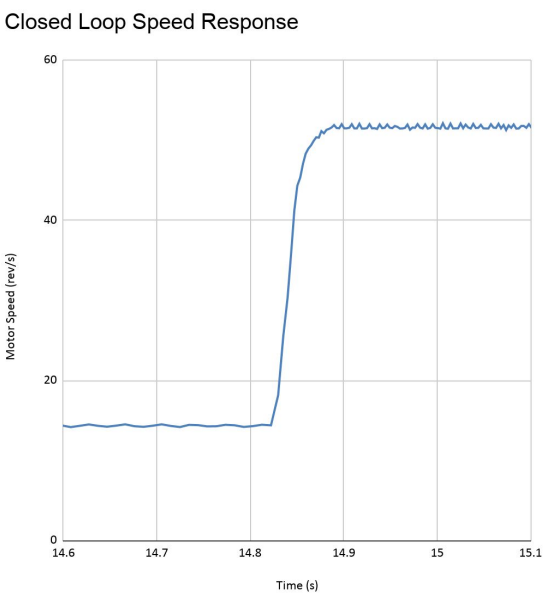
5.1 Methodology

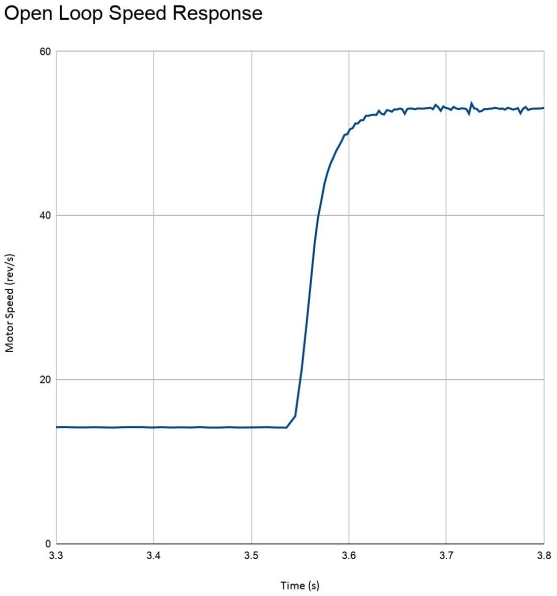
The test was run with both the PID system controlling the motor speed, and with the throttle manually set to a constant value. This control was achieved using the Quartus command line interface to set the corresponding registers.

Using the testbench described in the Tools section, the motor was taken between ~14 rev/sec and ~52 rev/sec, with speed sensor data collected using the UART speed output/UART TX components.

The time taken for the speed to reach 95% of the goal value was calculated and averaged over three trials for each response.

5.2 Results





95% Rise Time:

Closed loop: 0.047 seconds

Open loop: 0.070 seconds

Making the closed loop method 67% faster.

5.3 Discussion

The experiment does not effectively demonstrate the full performance improvement possible with a PID system in place, nor does it eliminate the possibility of the same improvements being applied to conventional sequential flight controller systems.

It demonstrates that performance improvements can be implemented within FPGA hardware that do not either decrease the performance or increase the complexity of the rest of the system. Unlike comparative sequential systems.

Despite the broad goals, the experiment still contains inaccuracies that, although judged have minimal effect on the conclusions, could still be significant.
When the PID module was used to control the speed, a target speed was established by reading back the speed sensor reading from the open loop trial. This method introduces slight differences between the current and target speeds in the open and closed loop. This can be observed in the data, the final speed for the open loop trial is perceptibly more than that of the closed loop (graphs above).
Other inaccuracies considered include, sample size, temperature variations, voltage variations, and enclosure movement.

6. Recommendations for Future Work

Even though all components were created. There still remains work that could significantly benefit the project. Some of these were previously discussed in their relevant discussion sections, but are summarised here:

Unfinished work

- PID still has some hardware bugs that affect its practicality
- Reduced resource utilisation necessary for further additions
- IMU not yet tested/integrated

Improvements to existing work

- The synchronization could be improved in some modules to reduce latencies
- PID module could be tuned much better to improve motor response speed and accuracy
- Dshot rewritten in a better coding style for easier debugging and modification

Continuations to existing work

- Expanding to multiple motors
- Increase data logging bandwidth

7. References

[1] V. Askue, “Fly-by-wire,” *Air Med. J.*, vol. 22, no. 6, pp. 4–5, Nov. 2003.

[2] “Betaflight Documentation Hardware Reference,” *GitHub*. [Online]. Available: https://github.com/betaflight/betaflight/wiki/Hardware-Reference. [Accessed: 24-Oct-2019].

[3] “Extended Kalman Filter Navigation Overview and Tuning — Dev documentation.” [Online]. Available: http://ardupilot.org/dev/docs/extended-kalman-filter.html. [Accessed: 31-Mar-2019].

[4] Intel, “MAX 10 FPGA Device Architecture,” p. 31.

[5] N. Monterrosa, J. Montoya, F. Jarquín, and C. Bran, “Design, development and implementation of a UAV flight controller based on a state machine approach using a FPGA embedded system,” in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–8.

[6] J. Young and A. R. Price, “FPGA based UAV flight controller,” in *Proceedings of the Eleventh Australian International Aerospace Congress*, 2005, pp. 1–5.

[7] G. Premkumar and R. Jayalakshmi, “Design and Implementation of FPGA Based Quadcopter,” vol. 5, no. 3, p. 5, Mar. 2018.

[8] C. Dominguez-Bonilla, A. Gutierrez, F. Jimenez, and H. R. Chamorro, “SysML methodology for FPGA-based Controller design for quadcopters,” in *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2016, pp. 1–6.

[9] J. C. Vianna Junior, J. C. De Paula, G. V. Leandro, and M. C. Bonfim, “Stability Control of a Quad-rotor Using a PID Controller,” *Braz. J. Instrum. Control*, vol. 1, no. 1, p. 15, Oct. 2013.

[10] Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong, China, H. L. Chan, and K. T. Woo, “Design and

Control of Small Quadcopter System with Motor Closed Loop Speed Control," *Int. J. Mech. Eng. Robot. Res.*, 2015.
[11] Z. Qun, H. Zhining, W. Hongshuo, L. Hailin, and Q. Zhaowei, "ARM (advanced RISC (reduced instruction set computer) machines) and FPGA (field-programmable gate array) based navigation and flight control system for unmanned helicopter," CN102360218 (A), 22-Feb-2012.
[12] "DSHOT600 & 150 -> digital one shot motor (ESC) protocol by blckmn · Pull Request #1282," *GitHub*. [Online]. Available: https://github.com/betaflight/betaflight/pull/1282. [Accessed: 31-May-2019].